Laravel 9 · Eloquent · Начало работы

Введение

Laravel содержит библиотеку Eloquent ORM (объектно-реляционное отображение), которая позволяет с удовольствием взаимодействовать с базой данных. При использовании Eloquent каждая таблица БД имеет соответствующую «Модель», которая используется для взаимодействия с этой таблицей. Помимо получения записей из таблицы БД, модели Eloquent также позволяют вставлять, обновлять и удалять записи из таблицы.

> {tip} Перед началом работы обязательно настройте соединение с БД в конфигурационном файле `config/database.php`. Для получения дополнительной информации о настройке БД ознакомьтесь с документацией по конфигурированию БД.

Генерация классов модели

Модели расширяют класс `Illuminate\Database\Eloquent\Model`. Чтобы сгенерировать новую модель Eloquent, используйте команду `make:model` Artisan. Эта команда поместит новый класс модели в каталог `app/Models` вашего приложения:

```
php artisan make:model Flight
```

При создании модели вы можете сгенерировать миграцию БД, используя параметр `--migration` или `-m`:

```
php artisan make:model Flight --migration
```

При создании модели вы можете попутно генерировать различные типы классов, например фабрики, наполнители, политики авторизации, контроллеры и запросы форм. Кроме того, эти параметры можно комбинировать для создания сразу нескольких классов:

```
# Создать модель и класс FlightFactory ...
php artisan make:model Flight --factory
```

```
php artisan make:model Flight -f

# Создать модель и класс FlightSeeder...
php artisan make:model Flight --seed
php artisan make:model Flight -s

# Создать модель и класс FlightController...
php artisan make:model Flight --controller
php artisan make:model Flight -c

# Создать модель, ресурсный контролер FlightController и запросы форм ...
php artisan make:model Flight --controller --resource --requests
php artisan make:model Flight -crR

# Создать модель и класс FlightPolicy ...
php artisan make:model Flight --policy

# Создать модель, миграцию, фабрику, наполнитель и контроллер ...
php artisan make:model Flight -mfsc

# Ярлык для создания модели, миграции, фабрики, наполнителя, политики, контроллера и запрос
php artisan make:model Flight --all

# Создать сводную модель...
php artisan make:model Member --pivot
```

Соглашения по именованию моделей Eloquent

Модели, созданные командой `make:model`, будут помещены в каталог `app/Models`. Давайте рассмотрим базовый класс модели и обсудим некоторые ключевые соглашения Eloquent:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

Именование таблиц

Взглянув на приведенный выше пример, вы могли заметить, что мы не сообщили Eloquent, какая таблица БД соответствует нашей модели `Flight`. По соглашению, в качестве имени таблицы будет использоваться имя класса в «змеином_регистре», во множественном числе, если явно не указано другое. В нашем случае, Eloquent будет предполагать, что модель `Flight` хранит записи в таблице `flights`, а модель `AirTrafficController` – в таблице `air_traffic_controllers`.

Если таблица БД вашей модели не соответствует этому соглашению, вы можете вручную указать имя таблицы модели, определив свойство `table` в модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Таблица БД, ассоциированная с моделью.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Первичные ключи

Eloquent также предполагает, что в соответствующей таблице БД каждой модели есть столбец первичного ключа с именем `id`. При необходимости вы можете определить защищенное свойство `$primaryKey` в модели, чтобы указать другой столбец, который служит первичным ключом:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Первичный ключ таблицы БД.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

Кроме того, Eloquent предполагает, что первичный ключ является автоинкрементным целочисленным значением, что означает, что Eloquent автоматически преобразует первичный ключ в целое число. Если вы хотите использовать неинкрементный или нечисловой первичный ключ, вы должны определить общедоступное свойство `$incrementing` в модели, для которого установлено значение `false`:

```
<?php

class Flight extends Model
{
    /**
     * Указывает, что идентификаторы модели являются автоинкрементными.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

Если первичный ключ модели не является целочисленным, то определите защищенное свойство `$keyType` в модели. Это свойство имеет значение типа `string`:

```
<?php

class Flight extends Model
{
    /**
     * Тип данных автоинкрементного идентификатора.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

«Составные» первичные ключи

Eloquent требует, чтобы каждая модель имела по крайней мере один однозначно идентифицирующий «ID», который может служить ее первичным ключом. «Составные» первичные ключи не поддерживаются моделями Eloquent. Однако вы можете добавить дополнительные многоколоночные уникальные индексы к таблицам базы данных в дополнение к однозначно определяющему (уникальному) первичному ключу таблицы.

Временные метки

По умолчанию Eloquent ожидает, что столбцы `created_at` и `updated_at` будут существовать в соответствующей таблице БД модели. Eloquent автоматически устанавливает значения этих столбцов при создании или обновлении моделей. Если вы не хотите, чтобы эти столбцы автоматически управлялись Eloquent, вы должны определить свойство `$timestamps` модели со значением `false`:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Следует ли обрабатывать временные метки модели.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

Если вам нужно настроить формат временных меток модели, то укажите необходимый формат для свойства `$dateFormat`. Это свойство определяет, как атрибуты даты хранятся в БД, а также их формат при сериализации модели в массив или JSON:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Формат хранения столбцов даты модели.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

Если вам нужно настроить имена столбцов, используемых для хранения временных меток, то укажите значения для констант `CREATED_AT` и `UPDATED_AT` в модели:

```php
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'updated_date';
}
```

Соединения с БД

По умолчанию все модели Eloquent будут использовать соединение с БД, настроенное для вашего приложения. Если вы хотите указать другое соединение, которое должно использоваться при взаимодействии с определенной моделью, вы должны определить свойство `$connection` модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Соединение с БД, которое должна использовать модель.
     *
     * @var string
     */
    protected $connection = 'sqlite';
}
```

Значения атрибутов по умолчанию

По умолчанию вновь созданный экземпляр модели не будет содержать никаких значений атрибутов. Если вы хотите определить значения по умолчанию для некоторых атрибутов модели, то укажите необходимые значения в свойстве `$attributes` модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Значения по умолчанию для атрибутов модели.
     *
     * @var array
     */
    protected $attributes = [
        'delayed' => false,
    ];
}
```

Получение моделей

Создав модель и связанную с ней таблицу БД, все готово к извлечению данных из БД. Вы должны думать о каждой модели Eloquent как о мощном построителе запросов, позволяющем свободно выполнять запросы к таблице БД, связанной с моделью. Метод модели `all` получит все записи из связанной с моделью таблицы БД:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

Создание запросов

Метод Eloquent `all` вернет все результаты из таблицы модели. Однако, поскольку каждая модель Eloquent служит построителем запросов, вы можете добавить дополнительные условия к запросам, а затем вызвать метод `get` для получения результатов:

```
$flights = Flight::where('active', 1)
               ->orderBy('name')
               ->take(10)
               ->get();
```

> {tip} Поскольку модель Eloquent является построителем запросов, вам следует просмотреть все методы, предлагаемые построителем запросов. Вы можете использовать любой из этих методов при написании запросов Eloquent.

Обновление моделей

Если у вас уже есть экземпляр модели Eloquent, полученный из БД, вы можете «обновить» модель, используя методы `fresh` и `refresh` . Метод `fresh` повторно извлечет модель из БД. Существующий экземпляр модели не будет затронут:

```
$flight = Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();
```

Метод `refresh` повторно обновит существующую модель, используя свежие данные из БД. Кроме того, будут обновлены все загруженные отношения:

```
$flight = Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"
```

Коллекции

Как мы видели, методы Eloquent, такие как `all` и `get`, получают несколько записей из БД. Однако эти методы не возвращают простой массив PHP. Вместо этого возвращается экземпляр `Illuminate\Database\Eloquent\Collection`.

Класс Eloquent `Collection` расширяет базовый класс Laravel `Illuminate\Support\Collection`, который содержит множество полезных методов для взаимодействия с коллекциями данных. Например, метод `reject` используется для удаления моделей из коллекции на основе результатов вызванного замыкания:

```
$flights = Flight::where('destination', 'Paris')->get();

$flights = $flights->reject(function ($flight) {
    return $flight->cancelled;
});
```

Помимо методов, предоставляемых базовым классом коллекции Laravel, класс коллекции Eloquent содержит несколько дополнительных методов, которые специально предназначены для взаимодействия с коллекциями моделей Eloquent.

Поскольку все коллекции Laravel реализуют итерируемые интерфейсы PHP, вы можете перебирать коллекции, как если бы они были массивом:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Разбиение результатов

Вашему приложению может не хватить памяти, если вы попытаетесь загрузить десятки тысяч записей Eloquent с помощью методов `all` или `get`. Вместо использования этих методов можно использовать метод `chunk` для более эффективной обработки большого количества моделей.

Метод `chunk` будет извлекать подмножество моделей Eloquent, передавая их в замыкание для обработки. Поскольку за один раз извлекается только текущая коллекция моделей Eloquent, метод `chunk` обеспечивает значительно меньшее потребление памяти при работе с большим количеством моделей:

```
use App\Models\Flight;

Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
        //
    }
});
```

Первым аргументом, передаваемым методу `chunk`, является количество записей, которые вы хотите получить за «порцию». Замыкание, переданное в качестве второго аргумента, будет вызвано для каждой части записей, полученной из БД. Будет выполнен запрос к БД для получения каждой части записей, переданных замыканию.

Если вы фильтруете результаты метода `chunk` на основе столбца, который вы также будете обновлять при итерации результатов, вам следует использовать метод `chunkById`. Использование метода `chunk` в этих сценариях может привести к неожиданным и противоречивым результатам. Внутренне метод `chunkById` всегда будет извлекать модели со столбцом `id`, большим, чем у последней модели в предыдущей «порции»:

```
Flight::where('departed', true)
    ->chunkById(200, function ($flights) {
        $flights->each->update(['departed' => false]);
    }, $column = 'id');
```

Разбиение результатов с отложенными коллекциями

Метод `lazy` работает аналогично методу `chunk` в том смысле, что он выполняет запрос по частям. Однако вместо передачи каждого фрагмента непосредственно в замыкание, метод `lazy()` возвращает экземпляр `LazyCollection` одноуровневых моделей Eloquent, что позволяет вам взаимодействовать с результатами как с единым потоком:

```
use App\Models\Flight;

foreach (Flight::lazy() as $flight) {
    //
}
```

Если вы фильтруете результаты метода `lazy` по столбцу, который впоследствии будет обновлен при итерации результатов, то вам следует использовать метод `lazyById`. Внутренне метод `lazyById` всегда будет извлекать модели со столбцом `id`, большим, чем у последней модели в предыдущей «порции»:

```
Flight::where('departed', true)
    ->lazyById(200, $column = 'id')
    ->each->update(['departed' => false]);
```

Вы можете фильтровать результаты по убыванию `id`, используя метод `lazyByIdDesc`.

Курсоры

Подобно методу `lazy`, метод `cursor` используется для значительного уменьшения потребления памяти вашим приложением при итерации десятков тысяч записей модели Eloquent.

Метод `cursor` выполнит только один запрос к БД; однако отдельные модели Eloquent не будут включены в результирующий набор, пока они не будут фактически итерированы. Следовательно, только одна модель Eloquent хранится в памяти в любой момент времени при итерации с использованием курсора.

> {note} Поскольку метод `cursor` всегда хранит в памяти только одну модель Eloquent, то нетерпеливая загрузка отношений недопустима. Если вам нужно нетерпеливо загрузить отношения, то рассмотрите возможность использования метода `lazy`.

Внутри метод `cursor` использует генераторы PHP для реализации этого функционала:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
    //
}
```

Курсор возвращает экземпляр `Illuminate\Support\LazyCollection`. Отложенные коллекции позволяют использовать многие методы коллекций, доступные в типичных коллекциях Laravel, при одновременной загрузке в память только одной модели:

```
use App\Models\User;

$users = User::cursor()->filter(function ($user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Хотя метод `cursor` использует гораздо меньше памяти, чем обычный запрос (удерживая в памяти только одну модель Eloquent), он все равно в конечном итоге может исчерпать память. Это связано с тем, что драйвер PDO PHP внутренне кэширует все необработанные результаты запросов в своем буфере. Если вы имеете дело с очень большим количеством записей Eloquent, то рассмотрите возможность использования метода `lazy`.

Расширенные подзапросы

Выборка

Eloquent также предлагает поддержку расширенных подзапросов, которая позволяет извлекать информацию из связанных таблиц в одном запросе. Например, давайте представим, что у нас есть таблица `destinations` (пункты назначения) и `flights` (рейсы). В таблице `flights` содержится столбец `arrived_at`, который указывает, когда рейс прибыл в пункт назначения.

Используя функциональность подзапроса, доступную для методов `select` и `addSelect` построителя запросов, мы можем выбрать все `destinations` и название рейса, который последним прибыл в этот пункт назначения, используя один запрос:

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
    ->limit(1)
])->get();
```

Сортировка

Кроме того, метод `orderBy` построителя запросов поддерживает подзапросы. Продолжая использовать наш пример полетов, мы можем использовать этот метод для сортировки всех пунктов назначения в зависимости от того, когда последний рейс прибыл в этот пункт назначения. Опять же, это можно сделать при выполнении одного запроса к БД:

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

Извлечение отдельных моделей

В дополнение к получению всех записей, соответствующих указанному запросу, вы также можете получить отдельные записи, используя методы `find`, `first` или `firstWhere`. Вместо того, чтобы возвращать коллекцию моделей, эти методы возвращают единственный экземпляр модели:

```
use App\Models\Flight;

// Получить модель по ее первичному ключу ...
$flight = Flight::find(1);

// Получить первую модель, соответствующую условиям запроса ...
$flight = Flight::where('active', 1)->first();

// Альтернатива извлечению первой модели, соответствующей условиям запроса ...
$flight = App\Models\Flight::firstWhere('active', 1);
```

По желанию можно выполнить какое-то другое действие, если результаты не найдены. Методы `findOr` и `firstOr` возвращают один экземпляр модели или, если результаты не найдены, выполняют указанное замыкание. Значение, возвращаемое замыканием, будет считаться результатом метода:

```
$flight = Flight::findOr(1, function () {
    // ...
});

$flight = Flight::where('legs', '>', 3)->firstOr(function () {
    // ...
});
```

Исключения при отсутствии результатов запроса

По желанию можно выбросить исключение, если модель не найдена. Это особенно полезно в маршрутах или контроллерах. Методы `findOrFail` и `firstOrFail` будут получать первый результат запроса; однако, если результат не найден, будет выброшено исключение `Illuminate\Database\Eloquent\ModelNotFoundException`:

```
$flight = Flight::findOrFail(1);

$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

Если исключение не перехвачено, то клиенту автоматически отправляется HTTP-ответ `404`:

```
use App\Models\Flight;

Route::get('/api/flights/{id}', function ($id) {
    return Flight::findOrFail($id);
});
```

Получение или создание моделей

Метод `firstOrCreate` попытается найти запись в БД, используя указанные пары столбец / значение. Если модель не может быть найдена в БД, будет вставлена запись с атрибутами, полученными в результате объединения первого аргумента массива с необязательным вторым аргументом массива:

Метод `firstOrNew`, как и `firstOrCreate`, попытается найти в БД запись, соответствующую указанным атрибутам. Однако, если модель не найдена, будет возвращен новый экземпляр модели. Обратите внимание, что модель, возвращенная `firstOrNew`, еще не сохранена в БД. Вам нужно будет вручную вызвать метод `save`, чтобы сохранить его:

```
use App\Models\Flight;
```

```
// Получить рейс по `name` или создать его, если его не существует ...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// Получить рейс по `name` или создать его с атрибутами `name`, `delayed` и
`arrival_time` ...
$flight = Flight::firstOrCreate(
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Получить рейс по `name` или создать новый экземпляр Flight ...
$flight = Flight::firstOrNew([
    'name' => 'London to Paris'
]);

// Получить рейс по `name` или создать экземпляр с атрибутами `name`, `delayed` и
`arrival_time` ...
$flight = Flight::firstOrNew(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

Извлечение Агрегатов

При взаимодействии с моделями Eloquent вы также можете использовать `count`, `sum`, `max` и другие агрегатные методы, предоставляемые построителем запросов Laravel. Как и следовало ожидать, эти методы возвращают соответствующее скалярное значение вместо экземпляра модели Eloquent:

```
$count = Flight::where('active', 1)->count();

$max = Flight::where('active', 1)->max('price');
```

Вставка и обновление моделей

Вставка

Конечно, при использовании Eloquent нам нужно не только извлекать модели из БД. Также нам нужно вставлять новые записи. К счастью, Eloquent делает это просто. Чтобы вставить новую запись в БД, вы должны создать экземпляр новой модели и установить атрибуты модели. Затем вызовите метод `save` экземпляра модели:

```
<?php

namespace App\Http\Controllers;
```

```
use App\Http\Controllers\Controller;
use App\Models\Flight;
use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * Сохранить новый рейс в базе данных.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        // Валидация запроса ...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}
```

В этом примере мы присваиваем параметр `name` из входящего HTTP-запроса атрибуту `name` экземпляра модели `App\Models\Flight`. Когда мы вызываем метод `save`, запись будет вставлена в БД. Временные метки `created_at` и `updated_at` будут автоматически установлены при вызове метода `save`, поэтому нет необходимости устанавливать их вручную.

В качестве альтернативы вы можете использовать метод `create`, чтобы «сохранить» новую модель с помощью одного оператора PHP. Вставленный экземпляр модели будет возвращен вам методом `create`:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

Однако, перед использованием метода `create` вам нужно будет указать свойство `fillable` или `guarded` в классе модели. Эти свойства необходимы, потому что все модели Eloquent по умолчанию защищены от уязвимостей массового присвоения. Чтобы узнать больше о массовом присвоении, обратитесь к документации.

Обновление

Метод `save` также используется для обновления моделей, которые уже существуют в БД. Чтобы обновить модель, вы должны извлечь ее и установить любые атрибуты, которые вы хотите обновить. Затем вы должны вызвать метод `save` . Опять же, временная метка `updated_at` будет автоматически обновлена, поэтому нет необходимости вручную устанавливать ее значение:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->name = 'Paris to London';

$flight->save();
```

Массовые обновления

Обновления также могут выполняться для моделей, соответствующих указанному запросу. В этом примере все рейсы, которые активны и имеют пункт назначения в Сан-Диего, будут помечены как задержанные:

```
Flight::where('active', 1)
      ->where('destination', 'San Diego')
      ->update(['delayed' => 1]);
```

Метод `update` ожидает массив пар ключей и значений, представляющих столбцы, которые должны быть обновлены. Метод `update` возвращает количество затронутых строк.

> {note} События модели Eloquent `saving` , `saved` , `updating` и `updated` при массовом обновлении **не будут инициированы** для затронутых моделей. Это связано с тем, что модели фактически никогда не извлекаются при массовом обновлении.

Изучение изменений атрибутов

Eloquent содержит методы `isDirty` , `isClean` и `wasChanged` для проверки внутреннего состояния модели и определения того, как изменились ее атрибуты с момента первоначального извлечения модели.

Метод `isDirty` определяет, были ли изменены какие-либо атрибуты модели с момента получения модели. Вы можете передать конкретное имя атрибута или массив атрибутов методу `isDirty` , чтобы определить, является ли какой-либо из атрибутов «грязным». Метод `isClean` определяет, остался ли атрибут неизменным с момента получения модели. Этот метод также принимает необязательный аргумент атрибута:

```
use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
```

```
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false
$user->isDirty(['first_name', 'title']); // true

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true
$user->isClean(['first_name', 'title']); // false

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

Метод `wasChanged` определяет, были ли изменены какие-либо атрибуты при последнем сохранении модели в текущем цикле запроса. При необходимости вы можете передать имя атрибута, чтобы увидеть, был ли изменен конкретный атрибут:

```
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged(['title', 'slug']); // true
$user->wasChanged('first_name'); // false
$user->wasChanged(['first_name', 'title']); // true
```

Метод `getOriginal` возвращает массив, содержащий исходные атрибуты модели, независимо от каких-либо изменений в модели с момента ее получения. При необходимости вы можете передать конкретное имя атрибута, чтобы получить исходное значение определенного атрибута:

```
$user = User::find(1);

$user->name; // John
$user->email; // john@example.com

$user->name = "Jack";
$user->name; // Jack
```

```
$user->getOriginal('name'); // John
$user->getOriginal(); // Массив исходных атрибутов ...
```

Массовое присвоение

Вы можете использовать метод `create`, чтобы «сохранить» новую модель с помощью одного оператора PHP. Вставленный экземпляр модели будет возвращен из метода:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

Однако перед использованием метода `create` вам нужно будет указать свойство `fillable` или `guarded` в классе модели. Эти свойства необходимы, потому что все модели Eloquent по умолчанию защищены от уязвимостей массового присвоения.

Уязвимость массового присвоения возникает, когда пользователь передает неожиданное поле HTTP-запроса, и это поле изменяет столбец в вашей базе данных, чего вы никак не ожидали. Например, злоумышленник может отправить параметр `is_admin` через HTTP-запрос, который затем передается методу `create` модели, позволяя пользователю перейти на уровень администратора.

Итак, для начала вы должны определить, какие атрибуты модели вы хотите сделать массово-назначаемыми. Вы можете сделать это используя свойство `$fillable` модели. Например, давайте сделаем атрибут `name` нашей модели `Flight` массово-назначаемым:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Атрибуты, для которых разрешено массовое присвоение значений.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

После того, как вы указали, какие атрибуты массово-назначаемые, вы можете использовать метод `create` для вставки новой записи в базу данных. Метод `create` возвращает вновь созданный экземпляр модели:

```
$flight = Flight::create(['name' => 'London to Paris']);
```

Если у вас уже есть экземпляр модели, вы можете использовать метод `fill` для заполнения его массивом атрибутов:

```
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

Массовое присвоение и JSON-столбцы

При назначении JSON-столбцов необходимо указать массово назначаемый ключ для каждого столбца в массиве `$fillable` модели. В целях безопасности Laravel не поддерживает обновление вложенных атрибутов JSON при использовании свойства `guarded`:

```
/**
 * Атрибуты, для которых разрешено массовое присвоение значений.
 *
 * @var array
 */
protected $fillable = [
    'options->enabled',
];
```

Защита массового присвоения

Если вы хотите, чтобы все ваши атрибуты были массово-назначаемыми, вы можете определить свойство модели `$guarded` как пустой массив. Если вы решите не защищать свою модель, вам следует позаботиться о том, чтобы всегда вручную обрабатывать массивы, переданные в методы Eloquent `fill`, `create` и `update`:

```
/**
 * Атрибуты, для которых НЕ разрешено массовое присвоение значений.
 *
 * @var array
 */
protected $guarded = [];
```

Обновления-вставки

Иногда требуется обновить существующую модель или создать новую, если подходящей модели не существует. Как и метод `firstOrCreate`, метод `updateOrCreate` сохраняет модель, поэтому нет необходимости вручную вызывать метод `save`.

В приведенном ниже примере, если существует рейс с пунктом отправления «Окленд» и пунктом назначения «Сан-Диего», его столбцы `price` и `discounted` будут обновлены. Если такой рейс не существует, то будет создан новый рейс с атрибутами, полученными в результате слияния первого массива аргументов со вторым массивом аргументов:

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

Если вы хотите выполнить несколько «обновлений-вставок» в одном запросе, вам следует использовать вместо этого метод `upsert`. Первый аргумент метода состоит из значений для вставки или обновления, а второй аргумент перечисляет столбцы, которые однозначно идентифицируют записи в связанной таблице. Третий и последний аргументы метода – это массив столбцов, которые следует обновить, если соответствующая запись уже существует в БД. Метод `upsert` автоматически устанавливает временные метки `created_at` и `updated_at`, если они разрешены в модели:

```
Flight::upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```

> {note} Все базы данных, кроме SQL Server, требуют, чтобы столбцы во втором аргументе метода `upsert` имели «первичный» или «уникальный» индекс. Вдобавок, драйвер базы данных MySQL игнорирует второй аргумент метода `upsert` и всегда использует «первичный» и «уникальный» индексы таблицы для обнаружения существующих записей.

Удаление моделей

Чтобы удалить модель, вызовите метод `delete` экземпляра модели:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->delete();
```

Вы можете вызвать метод `truncate`, чтобы удалить все записи базы данных, связанные с моделью. Операция `truncate` также сбрасывает все автоинкрементные идентификаторы в связанной с моделью таблице:

```
Flight::truncate();
```

Удаление существующей модели по ее первичному ключу

В приведенном выше примере мы извлекаем модель из БД перед вызовом метода `delete`. Однако если вы знаете первичный ключ модели, вы можете удалить модель, не извлекая ее, вызвав метод `destroy`. Помимо единственного первичного ключа, метод `destroy` может принимать несколько первичных ключей, массив первичных ключей или коллекцию первичных ключей:

```
Flight::destroy(1);

Flight::destroy(1, 2, 3);

Flight::destroy([1, 2, 3]);

Flight::destroy(collect([1, 2, 3]));
```

> {note} Метод `destroy` загружает каждую модель отдельно и вызывает для них метод `delete`, чтобы сработали события `deleting` и `deleted` должным образом для каждой модели.

Удаление моделей через запрос

Конечно, вы можете создать запрос Eloquent для удаления всех моделей, соответствующих условиям запроса. В этом примере мы удалим все рейсы, помеченные как неактивные. Как и массовые обновления, массовые удаления не вызывают никаких событий модели для удаляемых моделей:

```
$deleted = Flight::where('active', 0)->delete();
```

> {note} События модели Eloquent `deleting` и `deleted` при массовом удалении не будут инициированы для удаленных моделей. Это связано с тем, что модели фактически не извлекаются при выполнении оператора `delete`.

Программное удаление

Помимо фактического удаления записей из БД, Eloquent может выполнять «программное удаление» моделей. При таком удалении, они фактически не удаляются из БД. Вместо этого для каждой модели устанавливается атрибут `deleted_at`, указывающий дату и время, когда она была «удалена». Чтобы задействовать программное удаление, добавьте в модель трейт `Illuminate\Database\Eloquent\SoftDeletes`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
```

```
{
    use SoftDeletes;
}
```

> {tip} Трейт `SoftDeletes` автоматически типизирует атрибут `deleted_at` к экземпляру `DateTime / Carbon`.

Вам также следует добавить столбец `deleted_at` в таблицу БД. Построитель схемы Laravel содержит метод для создания этого столбца:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

Теперь, когда вы вызываете метод `delete` модели, в столбце `deleted_at` будут установлены текущие дата и время. Однако, запись в базе данных модели останется в таблице. При запросе модели, использующей программное удаление, программно удаленные модели будут автоматически исключены из всех результатов запроса.

Чтобы определить, был ли данный экземпляр модели программно удален, используйте метод `trashed`:

```
if ($flight->trashed()) {
    //
}
```

Восстановление программно удаленных моделей

По желанию можно «восстановить» программно удаленную модель. Чтобы восстановить такую модель, используйте метод `restore` экземпляра модели. Метод `restore` установит в столбце `deleted_at` модели значение `null`:

```
$flight->restore();
```

Вы также можете использовать метод `restore` в запросе для восстановления нескольких моделей. Опять же, как и другие «массовые» операции, это не вызовет никаких событий модели для восстанавливаемых моделей:

```
App\Models\Flight::withTrashed()
        ->where('airline_id', 1)
```

```
        ->restore();
```

Метод `restore` также используется при построении запросов, использующих отношения:

```
$flight->history()->restore();
```

Удаление моделей без возможности восстановления

По желанию можно действительно удалить модель из БД. Вы можете использовать метод `forceDelete`, чтобы окончательно удалить модель из таблицы БД:

```
$flight->forceDelete();
```

Вы также можете использовать метод `forceDelete` при построении запросов, использующих отношения Eloquent:

```
$flight->history()->forceDelete();
```

Запросы для моделей, использующих программное удаление

Включение программно удаленных моделей

Как отмечалось выше, программно удаленные модели будут автоматически исключены из результатов запроса. Однако, вы можете принудительно отобразить такие модели в результирующем наборе, используя метод `withTrashed` в запросе:

```
use App\Models\Flight;

$flights = Flight::withTrashed()
                ->where('account_id', 1)
                ->get();
```

Метод `withTrashed` также используется в запросе, использующем отношения:

```
$flight->history()->withTrashed()->get();
```

Извлечение только программно удаленных моделей

Метод `onlyTrashed` будет извлекать **только** программно удаленные модели:

```
$flights = Flight::onlyTrashed()
                ->where('airline_id', 1)
                ->get();
```

Очистка устаревших моделей

Иногда требуется периодически удалять ненужные модели. Для этого вы можете добавить трейт `Illuminate\Database\Eloquent\Prunable` или `Illuminate\Database\Eloquent\MassPrunable` к моделям, которые вы хотите периодически очищать. После добавления одного из трейтов к модели реализуйте метод `prunable`, который возвращает построитель запросов Eloquent, который разрешает модели, которые больше не нужны:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Flight extends Model
{
    use Prunable;

    /**
     * Получить запрос сокращаемой модели.
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function prunable()
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

Помечая модели как `Prunable` («сокращаемой»), вы также можете определить метод `pruning` модели для «очищения». Этот метод будет вызываться перед удалением модели. Этот метод может быть полезен для удаления любых дополнительных ресурсов, связанных с моделью, таких как сохраненные файлы, перед окончательным удалением модели из базы данных:

```
/**
 * Подготовить модель для очистки.
 *
 * @return void
 */
protected function pruning()
{
    //
}
```

После конфигурирования сокращаемой модели вы должны запланировать команду `model:prune` Artisan в классе `App\Console\Kernel` вашего приложения. Вы можете выбрать подходящий интервал выполнения этой команды:

```
/**
 * Определить расписание выполнения команд приложения.
 *
 * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
 * @return void
 */
protected function schedule(Schedule $schedule)
{
    $schedule->command('model:prune')->daily();
}
```

За кулисами команда `model:prune` автоматически обнаружит «сокращаемые» модели в каталоге `app/Models` вашего приложения. Если ваши модели находятся в другом месте, вы можете использовать параметр `--model` , чтобы указать имена классов моделей:

```
$schedule->command('model:prune', [
    '--model' => [Address::class, Flight::class],
])->daily();
```

Если вы хотите исключить определенные модели из числа моделей, обнаруженных для очистки, то вы можете использовать параметр `--except` :

```
$schedule->command('model:prune', [
    '--except' => [Address::class, Flight::class],
])->daily();
```

Вы можете сымитировать свой запрос очистки, выполнив команду `model:prune` с флагом `--pretend` . При имитации команда `model:prune` просто сообщит, сколько записей будет удалено, если команда действительно будет запущена:

```
php artisan model:prune --pretend
```

> {note} Программно удаляемые модели будут удалены (`forceDelete`) без возможности восстановления, если они соответствуют запросу сокращения.

Массовая очистка устаревших моделей

Когда модели имеют трейт `Illuminate\Database\Eloquent\MassPrunable` , то они удаляются из базы данных с помощью запросов массового удаления. Следовательно, не будет вызываться метод `pruning` , а также не будут инициированы события `deleting` и `deleted` моделей. Это связано с тем, что модели никогда не извлекаются перед удалением, что делает процесс очищения более эффективным:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\MassPrunable;

class Flight extends Model
{
    use MassPrunable;

    /**
     * Получить запрос сокращаемой модели.
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function prunable()
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

Репликация (тиражирование) моделей

Вы можете создать несохраненную копию существующего экземпляра модели, используя метод `replicate`. Этот метод особенно полезен, когда у вас есть экземпляры модели, которые имеют много одинаковых атрибутов:

```php
use App\Models\Address;

$shipping = Address::create([
    'type' => 'shipping',
    'line_1' => '123 Example Street',
    'city' => 'Victorville',
    'state' => 'CA',
    'postcode' => '90001',
]);

$billing = $shipping->replicate()->fill([
    'type' => 'billing'
]);

$billing->save();
```

Чтобы исключить репликацию одного или нескольких атрибутов в новую модель, можно передать массив методу `replicate`:

```php
$flight = Flight::create([
    'destination' => 'LAX',
    'origin' => 'LHR',
```

```
    'last_flown' => '2020-03-04 11:00:00',
    'last_pilot_id' => 747,
]);

$flight = $flight->replicate([
    'last_flown',
    'last_pilot_id'
]);
```

Области запроса

Глобальные области запроса

Глобальные области запроса позволяют добавлять ограничения ко всем запросам для конкретной модели. Программное удаление Laravel использует глобальные области запроса для извлечения «не удаленных» моделей из БД. Написание пользовательских глобальных областей предоставляет удобный и простой способ, гарантирующий, что в каждом запросе конкретной модели будут применены определенные ограничения.

Написание глобальных областей запроса

Написание глобальных областей запроса – это просто. Сначала определите класс, который реализует интерфейс `Illuminate\Database\Eloquent\Scope`. В Laravel нет специального каталога для размещения таких классов, поэтому вы можете разместить его в любом каталоге.

Интерфейс `Scope` требует, чтобы вы реализовали один метод: `apply`. В методе `apply` можно добавлять условия `where` или другие типы необходимых ограничений к запросу:

```
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
    /**
     * Применить область запроса к переданному построителю запросов.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $builder
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @return void
     */
    public function apply(Builder $builder, Model $model)
    {
        $builder->where('created_at', '<', now()->subYears(2000));
```

```
    }
}
```

> {tip} Если ваша глобальная область запроса добавляет столбцы в выражение `SELECT` запроса, то вы должны использовать метод `addSelect` вместо `select`. Это предотвратит непреднамеренную замену существующего выражения `SELECT` запроса.

Применение глобальных областей запроса

Чтобы назначить глобальную область запроса модели, вы должны переопределить метод модели `booted` и вызвать метод модели `addGlobalScope`. Метод `addGlobalScope` принимает экземпляр вашей области запроса как единственный аргумент:

```
<?php

namespace App\Models;

use App\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Выполнить все необходимые действия после загрузки модели.
     *
     * @return void
     */
    protected static function booted()
    {
        static::addGlobalScope(new AncientScope);
    }
}
```

После добавления области к модели `App\Models\User` в приведенном выше примере, вызов метода `User::all()` выполнит следующий SQL-запрос:

```
select * from `users` where `created_at` < 0021-02-18 00:00:00
```

Анонимные глобальные области запроса

Eloquent также позволяет вам определять глобальные области запроса с использованием замыканий, что особенно полезно для простейших областей запроса, которые не требуют отдельного класса. При определении глобальной области запроса с помощью замыкания вы должны указать желаемое имя области в качестве первого аргумента метода `addGlobalScope`:

```
<?php

namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Выполнить все необходимые действия после загрузки модели.
     *
     * @return void
     */
    protected static function booted()
    {
        static::addGlobalScope('ancient', function (Builder $builder) {
            $builder->where('created_at', '<', now()->subYears(2000));
        });
    }
}
```

Игнорирование глобальных областей запроса

Для исключения глобальной области в текущем запросе, используйте метод `withoutGlobalScope` . Этот метод принимает имя класса глобальной области в качестве единственного аргумента:

```
User::withoutGlobalScope(AncientScope::class)->get();
```

Или, если вы определили глобальную область с помощью замыкания, то вы должны передать строковое имя, которое вы присвоили глобальной области:

```
User::withoutGlobalScope('ancient')->get();
```

Если вы хотите удалить несколько или даже все глобальные области запроса, то вы можете использовать метод `withoutGlobalScopes` :

```
// Игнорировать все глобальные области запроса ...
User::withoutGlobalScopes()->get();

// Игнорировать некоторые глобальные области запроса ...
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

Локальные области запроса

Локальные области запроса позволяют определять общие наборы ограничений запросов, которые можно легко повторно использовать в приложении. Например, частое получение «популярных» пользователей. Чтобы определить область запроса, добавьте префикс `scope` к методу модели Eloquent.

Области запроса должны всегда возвращать экземпляр построителя запросов или `void` :

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Область запроса, содержащая только популярных пользователей.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    /**
     * Область запроса, содержащая только активных пользователей.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return void
     */
    public function scopeActive($query)
    {
        $query->where('active', 1);
    }
}
```

Использование локальных областей запроса

После определения области можно вызвать метод при выполнении запроса модели. Однако вы не должны включать префикс `scope` при вызове метода. Вы можете даже связывать вызовы различных областей:

```
use App\Models\User;

$users = User::popular()->active()->orderBy('created_at')->get();
```

Объединение нескольких областей модели Eloquent с помощью оператора `or` запроса может потребовать использования замыканий для достижения правильной логической группировки:

```
$users = User::popular()->orWhere(function (Builder $query) {
    $query->active();
})->get();
```

Поскольку это может быть громоздким, то Eloquent содержит метод `orWhere` «более высокого порядка», который позволяет вам свободно связывать области без использования замыканий:

```
$users = App\Models\User::popular()->orWhere->active()->get();
```

Динамические области запроса

По желанию можно определить область с параметрами. Для начала просто добавьте дополнительные параметры в сигнатуру метода области запроса. Параметры области должны быть определены после параметра `$query`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Область запроса, содержащая пользователей только определенного типа.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @param  mixed  $type
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
        return $query->where('type', $type);
    }
}
```

После того, как ожидаемые аргументы были добавлены в сигнатуру метода области запроса, вы можете передать аргументы при вызове области:

```
$users = User::ofType('admin')->get();
```

Сравнение моделей

По желанию можно определить, являются ли две модели «одинаковыми» или нет. Методы `is` и `isNot` могут использоваться для быстрой проверки наличия у двух моделей одного и того же первичного ключа, таблицы и соединения с базой данных:

```
if ($post->is($anotherPost)) {
    //
}

if ($post->isNot($anotherPost)) {
    //
}
```

Методы `is` и `isNot` также доступны при использовании отношений `belongsTo`, `hasOne`, `morphTo` и `morphOne`. Этот метод особенно полезен, если вы хотите сравнить связанную модель без запроса на получение этой модели:

```
if ($post->author()->is($user)) {
    //
}
```

События

> {tip} Хотите транслировать события Eloquent прямо в клиентское приложение? Посмотрите трансляцию событий модели Laravel.

Модели Eloquent инициируют некоторые события, что позволяет использовать следующие хуки жизненного цикла модели: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `trashed`, `forceDeleted`, `restoring`, `restored` и `replicating`.

Событие `retrieved` сработает, когда существующая модель будет извлечена из БД. Когда новая модель сохраняется в первый раз, инициируются события `creating` и `created`. События `updating` / `updated` будут инициированы при изменении существующей модели и вызове метода `save`. События `saving` / `saved` будут инициированы при создании или обновлении модели – даже если атрибуты модели не были изменены. События, оканчивающиеся на `-ed`, инициируются после сохранения изменений в модели.

Чтобы начать прослушивание событий модели, определите свойство `$dispatchesEvents` в модели Eloquent. Это свойство сопоставляет различные хуки жизненного цикла модели Eloquent с вашими собственными классами событий. Каждый класс событий модели должен ожидать получения экземпляра затронутой модели через свой конструктор:

```
<?php

namespace App\Models;

use App\Events\UserDeleted;
```

```
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Карта событий для модели.
     *
     * @var array
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

После определения и сопоставления событий вы можете использовать слушателей событий для их обработки.

> {note} События модели Eloquent `saved`, `updated`, `deleting` и `deleted` при массовом обновлении или удалении **не будут инициированы** для затронутых моделей. Это связано с тем, что модели фактически не извлекаются при массовом обновлении или удалении.

Использование замыканий

Вместо использования пользовательских классов событий можно регистрировать замыкания, которые выполняются при инициировании различных событий модели. Как правило, вы должны зарегистрировать эти замыкания в методе `booted` модели:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Выполнить все необходимые действия после загрузки модели.
     *
     * @return void
     */
    protected static function booted()
    {
        static::created(function ($user) {
            //
        });
    }
}
```

При необходимости вы можете использовать очереди анонимных слушателей событий при регистрации событий модели. Это укажет Laravel выполнить слушателя событий модели в фоновом режиме, используя очередь вашего приложения:

```
use function Illuminate\Events\queueable;

static::created(queueable(function ($user) {
    //
}));
```

Наблюдатели

Определение наблюдателей

Если прослушивается множество событий в модели, то можно использовать наблюдателей, чтобы сгруппировать пользовательских слушателей в одном классе. Классы наблюдателей имеют имена методов, созвучные событиям Eloquent, которые необходимо прослушивать. Каждый из этих методов получает затронутую модель в качестве единственного аргумента. Чтобы сгенерировать нового наблюдателя, используйте команду `make:observer` Artisan:

```
php artisan make:observer UserObserver --model=User
```

Эта команда поместит новый класс наблюдателя в каталог `app/Observers` вашего приложения. Если этот каталог не существует в вашем приложении, то Laravel предварительно создаст его. Созданный наблюдатель может выглядеть следующим образом:

```
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * Обработать событие «created» модели User.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }

    /**
     * Обработать событие «updated» модели User.
     *
     * @param  \App\Models\User  $user
```

```
     * @return void
     */
    public function updated(User $user)
    {
        //
    }

    /**
     * Обработать событие «deleted» модели User.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function deleted(User $user)
    {
        //
    }

    /**
     * Обработать событие «restored» модели User.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function restored(User $user)
    {
        //
    }

    /**
     * Обработать событие «forceDeleted» модели User.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function forceDeleted(User $user)
    {
        //
    }
}
```

Чтобы зарегистрировать наблюдателя, вам нужно вызвать метод `observe` наблюдаемой модели. Как правило, регистрация наблюдателей осуществляется в методе `boot` поставщика `App\Providers\EventServiceProvider`:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * Зарегистрировать любые события приложения.
 *
 * @return void
 */
public function boot()
```

```
{
    User::observe(UserObserver::class);
}
```

Кроме того, вы можете перечислить своих наблюдателей в свойстве `$observers` класса `App\Providers\EventServiceProvider` вашего приложения:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * Наблюдатели моделей вашего приложения.
 *
 * @var array
 */
protected $observers = [
    User::class => [UserObserver::class],
];
```

> {tip} Наблюдатель может прослушивать дополнительные события, такие как `saving` и `retrieved` . Эти события описаны в документации событий.

Наблюдатели и транзакции базы данных

Когда модели создаются в рамках транзакции базы данных, можно дать команду наблюдателю выполнять его обработчики событий только после того, как транзакция базы данных будет зафиксирована. Это достижимо путем определения свойства `$afterCommit` наблюдателя. При отсутствии транзакции, обработчики событий будут выполнены незамедлительно:

```
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * Обрабатывать события после фиксирования всех транзакций.
     *
     * @var bool
     */
    public $afterCommit = true;

    /**
     * Обработать событие «created» модели User.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function created(User $user)
```

```
    {
        //
    }
}
```

Подавление событий

По желанию можно временно «заглушить» все события, запускаемые моделью. Вы можете добиться этого, используя метод `withoutEvents`. Метод `withoutEvents` принимает замыкание как единственный аргумент. Любой код, выполняемый в этом замыкании, не будет запускать события модели и любое значение, возвращаемое замыканием, будет возвращено методом `withoutEvents`:

```
use App\Models\User;

$user = User::withoutEvents(function () use () {
    User::findOrFail(1)->delete();

    return User::find(2);
});
```

Тихое сохранение одной модели

По желанию можно «сохранить» конкретную модель, не вызывая никаких событий. Вы можете сделать это с помощью метода `saveQuietly`:

```
$user = User::findOrFail(1);

$user->name = 'Victoria Faith';

$user->saveQuietly();
```